(12) UK Patent Application (19) GB (11) 2 377 045 (13) A

(43) Date of A Publication 31.12.2002

(21) Application No 0115891.4

(22) Date of Filing 28.06.2001

(71) Applicant(s)
Sony Service Centre (Europe) N.V.
(Incorporated in Belgium)
Technologielaan 7, 1840 Londerzeel,
Belgium

(72) Inventor(s)
Jan Weytjens
Koen de Vroede
Tom Cool

(74) Agent and/or Address for Service
J A Kemp & Co.
14 South Square, Gray's Inn, LONDON,
WC1R 5JJ, United Kingdom

(51) INT CL[7]
G06F 9/445 9/44

(52) UK CL (Edition T)
G4A AFL APL

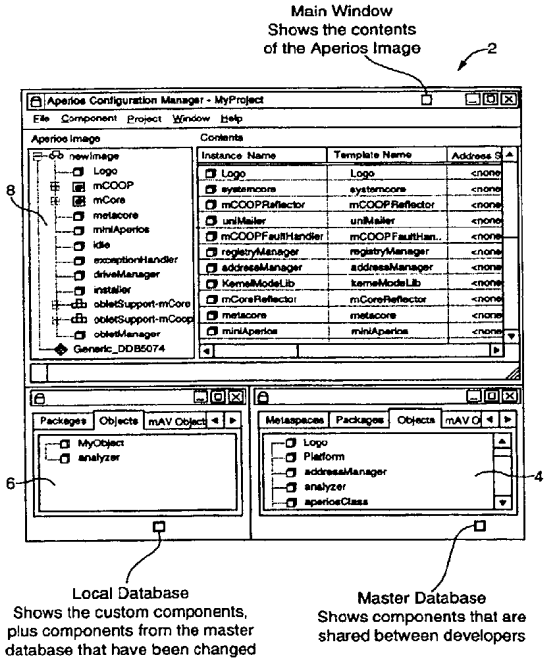
(56) Documents Cited
US 6223342 A **US 5845119 A**
A CASE-oriented configuration management agent, Tuppa & Selberherr, IASTED International Conference, Honolulu, 19-21 Aug 1996, pages 368-371, ISBN 0-88986-211-7

(58) Field of Search
UK CL (Edition T) **G4A** AFL APL
INT CL[7] **G06F** 9/44 9/445
Other: **Online: EPODOC, WPI,JAPIO, INSPEC**

(54) Abstract Title
Configuration manager

(57) A configuration manager for use in forming an image from a plurality of components including objects and collections of objects, the configuration manager including a selector for selecting components from a database for use in the image and an analyser for analysing the selected components to determine what other components are required by the selected components and means for ensuring that all components required by the selected components are themselves selected. Each component being stored in association with a plurality of attributes which at least indicate any other components required by the component. An editor may be provided for editing or modifying the selected components wherein the editor may amend the attributes stored in association with the components such that properties of a component can be changed without the need for recompilation. The configuration manager may further include a version controller for determining the version of components and converting the contents of older version components to the current version. The configuration manager may be implemented as a four layer structure comprising a persistence layer, a common code base layer, a layer of platform independent core classes and a platform specific GUI layer.

Fig.1.



GB 2 377 045 A

Fig.1.

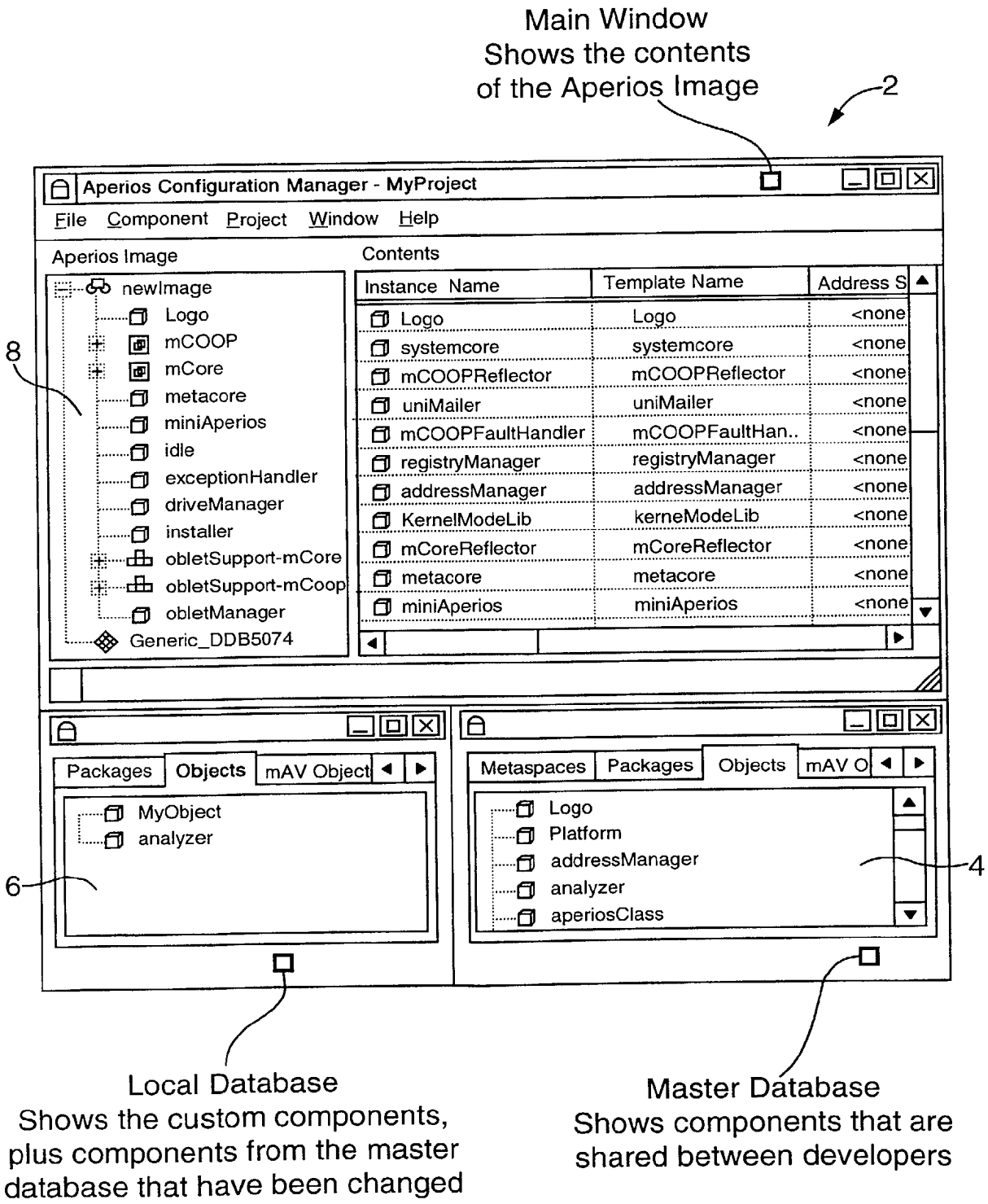

Main Window
Shows the contents
of the Aperios Image
2
Aperios Configuration Manager - MyProject
File Component Project Window Help
Aperios Image
Contents
newImage
Logo
mCOOP
mCore
metacore
miniAperios
idle
exceptionHandler
driveManager
installer
obletSupport-mCore
obletSupport-mCoop
obletManager
Generic_DDB5074
Instance Name
Template Name
Address S
Logo
systemcore
mCOOPReflector
uniMailer
mCOOPFaultHandler
registryManager
addressManager
KernelModeLib
mCoreReflector
metacore
miniAperios
<none
8
Packages
Objects
mAV Object
MyObject
analyzer
6
Metaspaces
Packages
Objects
mAV O
Logo
Platform
addressManager
analyzer
aperiosClass
4
Local Database
Shows the custom components,
plus components from the master
database that have been changed
Master Database
Shows components that are
shared between developers

Fig.2.

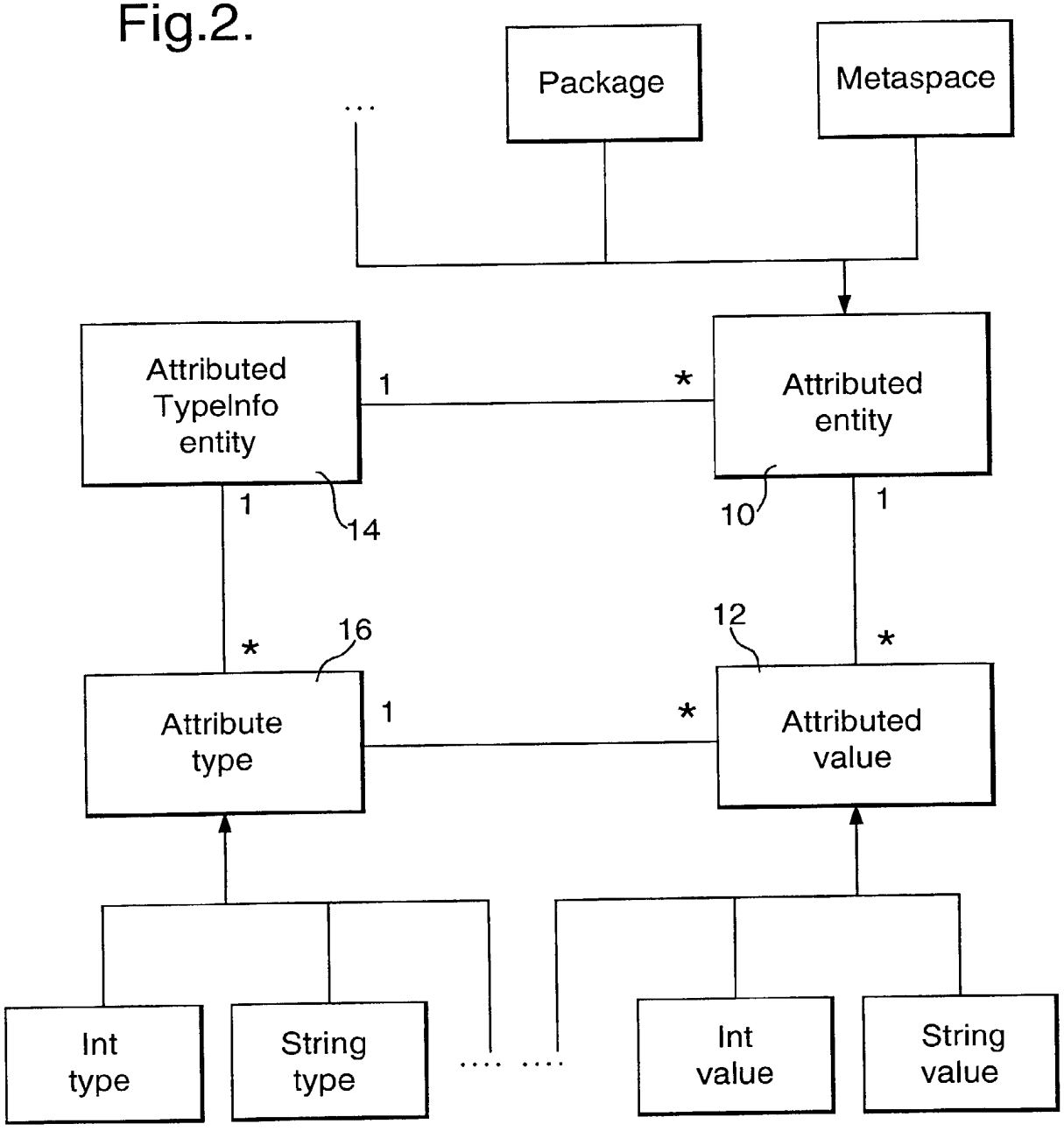

Package
Metaspace
...
Attributed TypeInfo entity
1
*
Attributed entity
1
14
10
1
*
16
12
*
Attribute type
1
*
Attributed value
Int type
String type
....
Int value
String value

Fig.3.

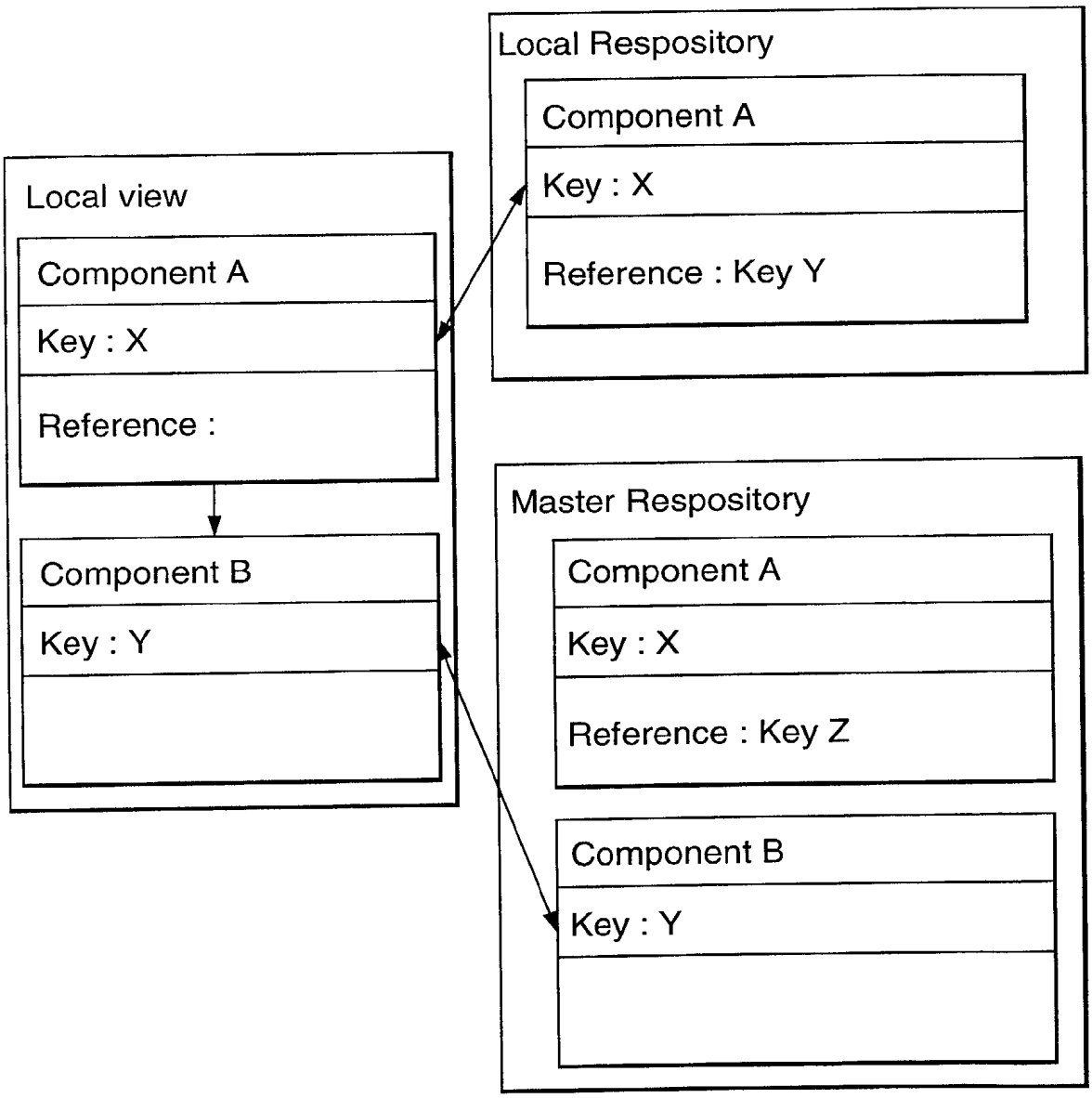

Local Respository
Component A
Key : X
Reference : Key Y
Local view
Component A
Key : X
Reference :
Component B
Key : Y
Master Respository
Component A
Key : X
Reference : Key Z
Component B
Key : Y

Fig.4.

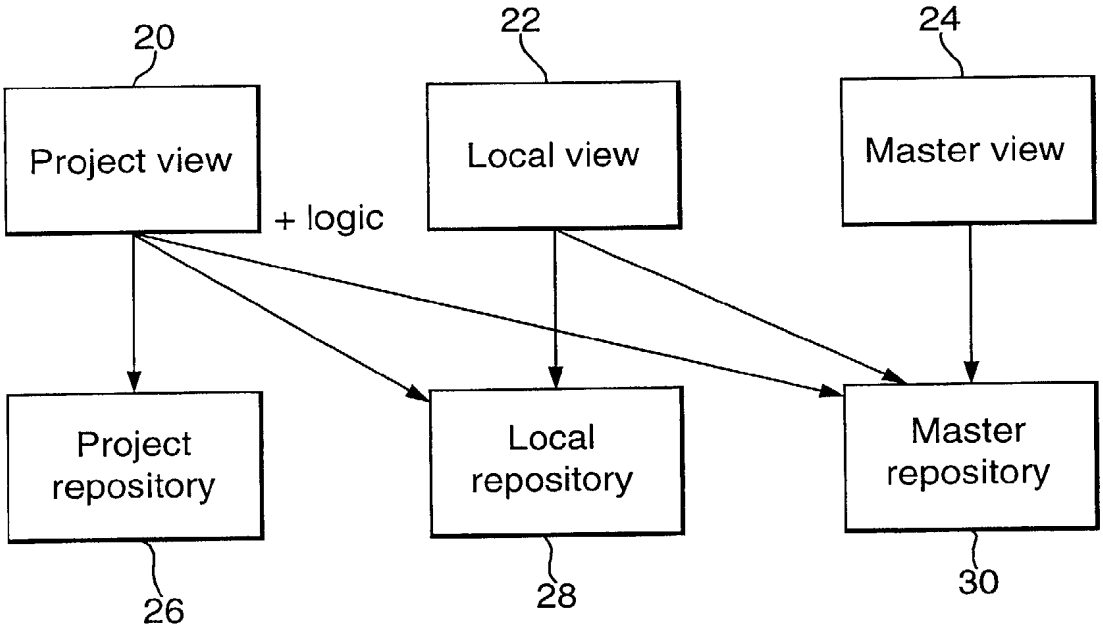

20
22
24
Project view
+ logic
Local view
Master view
Project repository
Local repository
Master repository
26
28
30

Fig.5.

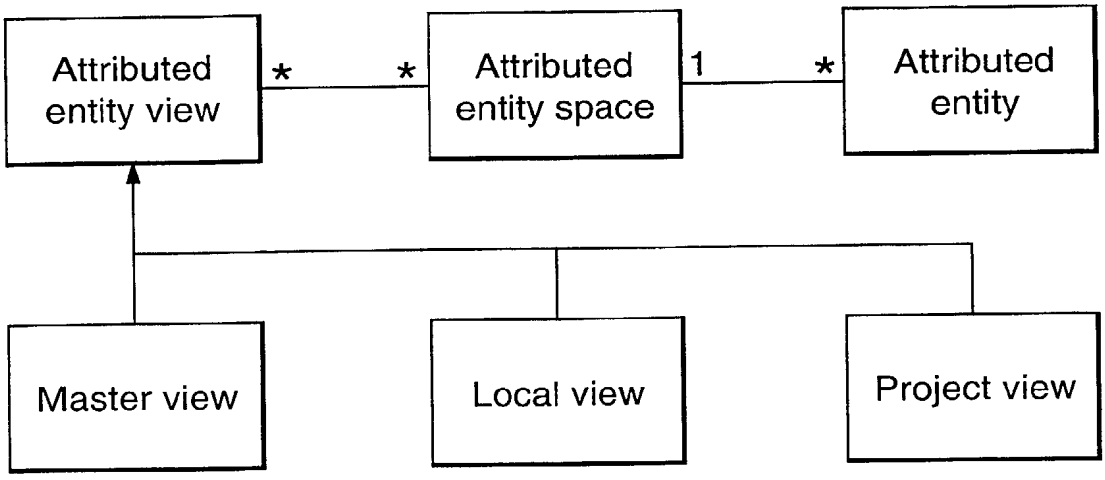

Attributed entity view
*
*
Attributed entity space
1
*
Attributed entity
Master view
Local view
Project view

Fig.6.

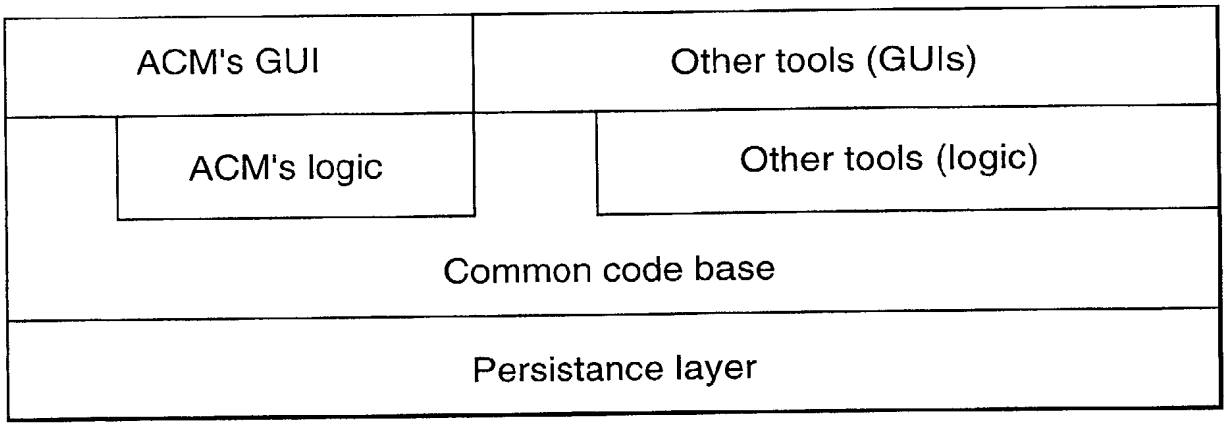
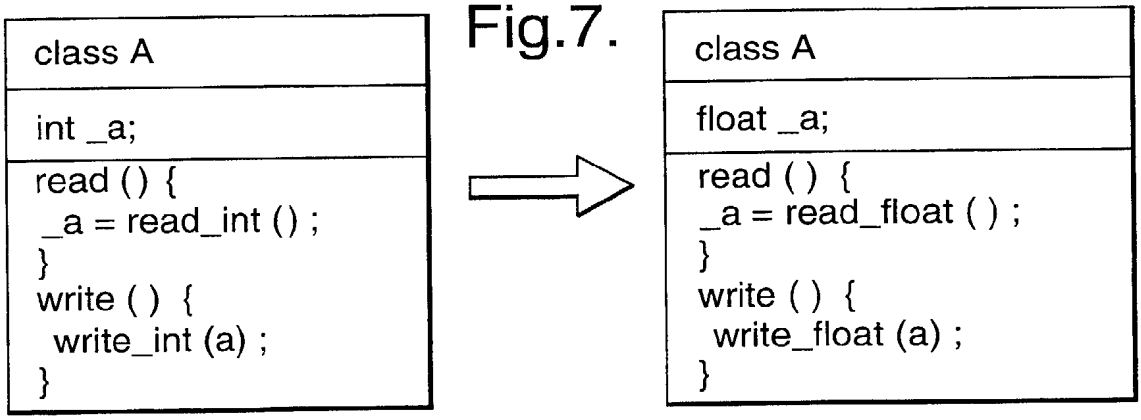


Fig.7.



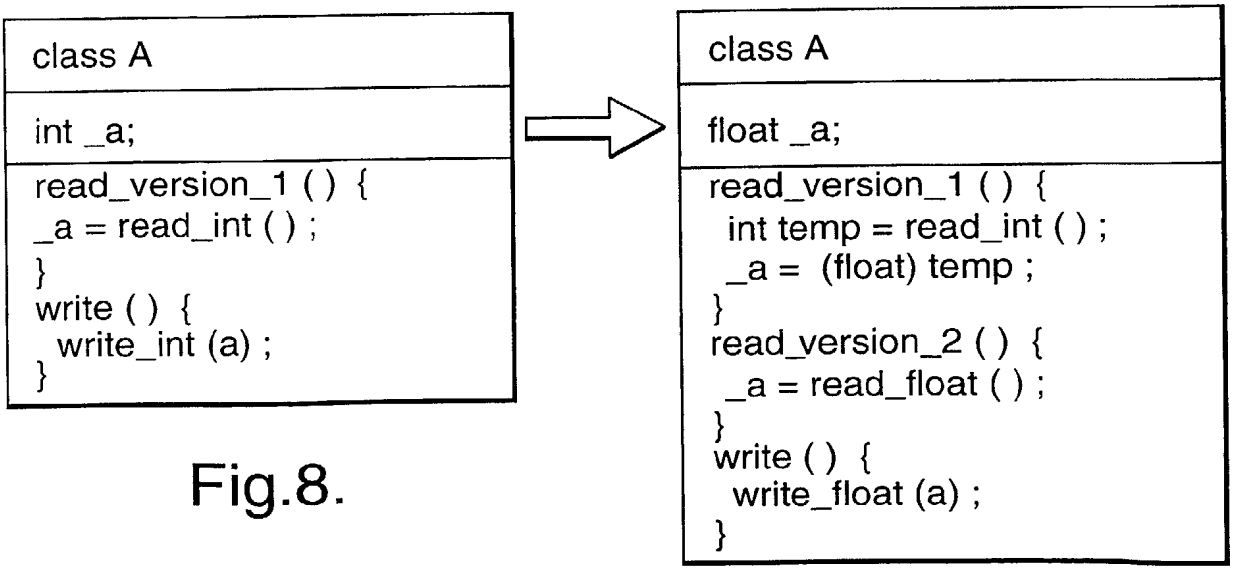


Fig.8.

CONFIGURATION MANAGER

The present application relates to a configuration manager, in particular, to a configuration manager for use in forming an image from a plurality of components.

Various object orientated programming systems are known in which binary images for running an application are constructed from a number of objects. Many different types of object may exist, together with collections of objects that provide operating system services to application objects and also collections of semantically related objects. These objects and groupings may be described as components of an image.

Object orientated programming is a very effective way of programming. However, considerable burden is placed on the programmer to provide a group of components which are correctly inter related.

The present application recognises for the first time the need for simplifying the process of assembling components of an image and facilitating the sharing of components between different users.

According to the present invention, there is provided a configuration manager for use in forming an image from a plurality of components including objects and metaspaces, the configuration manager including:

a selector for selecting components from a database for use in the image;

an analyser for analysing selected components to determine what other components are required by said selected components; and

means for ensuring that all components required by the selected components are themselves selected.

According to the present invention, there is also provided a method of forming an image from a plurality of components including objects and metaspaces, the method including:

selecting components from a database for use in the image;

analysing selected components to determine what other components are required by said selected components; and

ensuring that all components required by the selected component are themselves selected.

In this way, application developers are able to focus on their application's components, without worrying about what is required by the operating system or other components in the image of their application under construction.

Preferably, the means for or step of ensuring also ensures that all the components are properly configured.

Preferably, each component is stored in association with a plurality of attributes, the attributes at least indicating any other components required respectively by said each component and the analyser referring to the attributes.

In this way, when the analyser refers to the attributes of a particular component, it is able to determine from those attributes any other components required by that particular component. Hence, the configuration manager is able easily to determine all of the inter dependencies of the selected components.

The configuration manager may prompt a user to select other components required by the selected components or automatically select the other components required by the selected components.

In this way, an application developer can concentrate on selecting the important or significant components of the image under construction without monitoring which components are required to accompany the selected components.

An editor may be provided for editing selected components and storing the edited components separately to the database.

In particular, a project file may be provided for recording the selected components. Unmodified components from the database may be recorded merely by way of pointers to the database, whereas edited components may themselves be stored.

The editor may amend the attributes stored in association with the components.

In this way, the properties of a component can be changed without the need for recompilation. If the editor is used to change the dependency of a component, the new modified component will be stored in the project file. The configuration

manager can then determine what new components are therefore required for selection.

Preferably the configuration manager includes a display for displaying all of the components available from the database for selection and all of the components selected for the image, for instance those recorded for the project file.

Two views may be provided, one for a master database and one for the selected components required for the project image. Components of the master database selected for the project image may be illustrated in the view for the project image even though a pointer is merely provided to the master database. On the other hand, components which have been modified and stored in the view for the project image may be marked or highlighted so as to indicate that they have been modified.

Preferably, the configuration manager includes a version controller for determining the version of the components of the master database and for converting older version components to the current version and storing the converted components.

The configuration manager may be implemented as a four layer structure including a persistence layer, a common code base, a layer of platform independent core classes and a platform specific graphical user interface layer.

In order to provide robust persistence, the version information is implemented in the persistence layer.

The attributes are implemented in the common code base and any logic, such as the means for ensuring, is contained in the layer of platform independent core classes.

A system may be provided further comprising a compiler for compiling all of the selected components together to form the image. Hence, the method may further include compiling all of the selected components together to form the image.

According to the present invention, there is also provided a computer readable storage medium having recorded thereon code components that, when loaded on a computer and executed, will cause that computer to operate as defined above.

The invention will be more clearly understood from the following description given by way of example only, with reference to the accompanying drawings, in which:

Figure 1 illustrates a configuration manager display;

Figure 2 illustrates an attributed entity diagram;

Figure 3 illustrates a local view accessing local and master repositories;

Figure 4 illustrates a hierarchial viewing of repositories;

Figure 5 illustrates an attributed entity view diagram;

Figure 6 illustrates the four layer structure of a configuration manager;

Figure 7 illustrates an example of a definition change; and

Figure 8 illustrates a version dependent read function.

Applications that run on object oriented systems consist of carefully assembled collections of active objects. In embedded systems, these objects co-exist with the objects required by the operating system and are in a binary file called an image.

Such images may be run on computers, but will often be required by electronic devices or embedded devices which run the application. In these cases, the image has to be created in its entirety before being loaded onto the device. Since, in most cases, the image cannot be altered, it is important that it is correctly configured before it is loaded.

To create an image, it is proposed to provide a configuration manager which allows developers to easily define, configure and manage objects in images.

Several types of object can exist. It is also proposed to provide two types of object groupings, namely collections of objects that provide operating system services to application objects to be called metaspaces and collections of semantically related objects to be called packages. These objects and groupings will be described as components of an image.

The configuration manager may be provided with a display 2 as illustrated in Figure 1. The display 2 may be the display screen of a computer system in which the configuration manger is embodied.

The illustrated display is divided into three views 4, 6 and 8 representing respectively a master database, a local database and a project image. It is not necessary to provide more than the master database view 4 and the project image view 8, but, on the other hand, a number of different local database views could be provided.

The master database view 4 provides a representation of all of the components available from a master database to which the configuration manager has access. This master database should include all of the basic components of the object orientated operating system for which an image is being constructed. However, it may also include many other components which have been written and which have become available for general use. In order to construct an image, a user merely selects required components from the master database. In order to do this, in one embodiment, a user could merely drag the representation of a component in the master database view 4 across into the project image view 8. All of the selected components are then illustrated in the project image display 8.

Of course, since the configuration manager has selected one of the components already stored in the master database, it is not necessary for the data of that component to be duplicated in a portion of the memory allocated to the project image. Thus, the configuration manager can record that component by way of a reference to the appropriate component even though the project view 8 shows the component in question.

Of course, custom components can also be written for the application and these can be recorded as part of the project image or a local database.

The master database can be used as a read-only database and contain components that are shared by a development team. In a preferred embodiment, special administrator rights would be required to replace or add components to the master database.

On the other hand, the local database could be used to store an individual's own components or components whether they are original components from the master database that have been modified. Of course, this database could also be shared with other developers.

Thus, where a custom component or a component of the master database that has been modified is selected for the project image or the local database, that component is stored in the memory for the project image or the local database. Otherwise, where the component of the project image is merely a component from the local database or the master database, the configuration manager need only keep the reference to the appropriate component in the master database.

Preferably, each component in the image has properties that are configurable such as PIC/PID, stack and heap size, scheduling priorities and so on. The developer can create custom properties and change variables for each component. Preferably, the properties of the target platform can also be specified.

Thus, each component of the image preferably has configurable properties, for instance the metaspace on which an object runs or the memory areas where its code and data are to be loaded. The properties that characterize different types of components can change, as well as the default values of these properties. Indeed, due to changing requirements, the type of property may also be changed, for instance from an integer value to a string value.

The properties could be modelled using class data members. However, modification of the set of data members or the data members themselves (for example their default values) would then require recompilation depending on the implementation language. This is the case for C ++. It is here recognised that such changes should not require recompilation of the configuration manager and that this requirement should apply to the properties of all components in an image, including packages and metaspaces.

In view of the above, it is now proposed that each component has associated with it a plurality of attributes for modelling the properties of component.

It is proposed that properties should be modelled with typed attributes, that is, with pairs of classes; classes that implement named types and classes that implement variables corresponding to the type. Both the type and value classes derive respectively from an *AttributeType* and *AttributeValue* base class. The attribute value class, as the name suggests represents the value of the property. The type class

encapsulates type-information about the property (for example, the default value) and type specific constraints (for example, a maximum value for an integer).

Referring to Figure 2, image components are modelled by derivation from the *AttributedEntity* base class 10 because the properties (typed attributes) must be allowed to change. This base class is, in essence, a polymorphic container of instances of *AttributeValue*-derived classes 12. The attribute types for each component type are grouped together in an instance of the *AttributedEntityTypeInfo* class 14. This class, similar to the *AttributedEntity* class 10, maintains a list of polymorphic pointers to *AttributeType*-derived class instances 16, and can be considered as the meta data of the associated component type.

Some of the attributes allow the configuration manager to analyse the components and to determine what other components are required by them. In particular, the attributes of a component will indicate other components required by that component. Thus, for example, where an object requires a particular metaspace from the operating system, this will be indicated by way of the attributes.

Other attributes may be provided for other purposes such as influencing how the image is built, how the image will behave at runtime, etc.

The configuration manager can then ensure that all of those components have been selected, either by prompting the user to select them or by automatically selecting them. The additional components having been selected to this way are then of course themselves checked such that any components required by them are also selected.

It will be appreciated that, with an arrangement as discussed above, it is possible for a number of users to share the master database and each modify components in different ways according to their needs. To allow collaboration and sharing of components between developers, it is proposed that repositories should be available to permit concurrent access to predefined components. It is proposed that users can modify properties of these components as discussed above without effecting the work of others.

In order to do this, the master, local and project databases discussed above for the configuration manager can be generalised to a plurality of memories such as a

master repository, a local repository and a project repository. The master repository is read-only and can be shared by multiple users. The two other repositories are writable. The local repository stores modified versions of items in the master repository as well as new items. It can be shared by different users and projects.

For the configuration manager, the project repository stores modified components that are only relevant to the current project as well as settings necessary to build the image (for example the interrupt handling policy).

In order to identify a particular component, each component is stored in a repository based on a unique key. Thus, modified versions of a component always have the same key as the original, but are stored in a different repository. For example, when a component in the master database/repository is modified, the modified version is stored in the local or project repository using the same key. This is illustrated in Figure 3. In particular, the original component A, having key X, is stored in the master repository with reference to key Z. However, this component has been modified to include a reference to key Y and has been stored in the local repository, but still with key X. The local view of the configuration manager requiring a component with key X takes that component from the local repository. Since that component requires reference to key Y, the configuration manager therefore looks for a component having key Y. Since the local repository does not contain such a component, the local view refers to component B of the master repository. Thus, it will be seen that a component refers to another component by specifying the key of the referred component. This means that it is not possible for a component to refer to a certain version of another component explicitly. Version is determined by the context in which the reference is evaluated. Browsing the master repository using a master view displays the version that is stored in the master repository, but browsing the local view displays, in order of precedence, the version stored in the local repository or, if this does not exist, the version stored in the master repository.

Thus, as illustrated in Figure 4, this process of evaluating a reference in a context is encapsulated in views. A view is responsible for selecting a certain version of a component, based on a key or other form of ID. It specifically uses one

or more repositories using some order of precedence or hierarchy between those repositories to arrive at the correct version.

Thus, the project view 20 of Figure 4 would first look for a component in the project repository 26, then the local repository 28 and finally the master repository 30. On the other hand, the local view 22 would look for a component first in the local repository 28 and then the master repository 30 and the master view 24 would merely look in the master repository 30. Of course, the particular hierarchy assigned to the repositories can be different for each view. Thus, another project view might assign a hierarchy with the local repository as first choice, its own project repository as second choice, another project repository as third choice and the master repository as last choice.

This system is applicable to any arrangement in which different versions of the same type of data or file are stored in different repositories. As indicated previously, for the configuration manager discussed above, the graphical user interface (GUI) uses three repositories and three views. There is a view to browse the master database, a view to browse the local database and a view to browse the project repository. Of course, the project view is more complicated, since it also has to take into account the project's image composition which allows the user to explicitly select a certain version of a component to be included in the image.

Component versions are modelled by the configuration manager as instances of the *AttributedEntity* class. This is illustrated in Figure 5.

Repositories are instances of the *AttributedEntitySpace* class, which is a container for *AttributedEntities*. Finally, the view is an instance of an *AttributedEntityView* - derived class, implemented the specific mechanism to resolve references.

The configuration manager is preferably implemented as a four layer structure as shown in Figure 6. In particular, it has 1) a thin persistence layer, 2) a common code base of platform-independent core classes 3) a layer of platform-independent ACM-specific classes and 4) a platform-specific graphical user interface (GUI) layer.

The persistence layer can be used to particular advantage with the configuration manager.

Persistence refers to an object's ability to transcend time or space. A persistent object saves its state in a permanent storage system to make it possible for the process that created the object to terminate, without losing the information represented by the object. Later, the object may be reconstructed by another process and will behave in exactly the same way as it did in the initial process. However, the mere ability to make objects persistent is not sufficient to implement robust persistence, namely persistence that is resilient to changes in the definitions of the persistent classes. Such changes may for instances be required to accommodate design improvements. Components may be designed which are intended for use with the components of a particular master database but will not operate with the components of an earlier equivalent master database.

Changes may give rise to sychronisation problems, for instance class definitions compiled into the program may differ from definitions used by the data files. The result of this is that files in which instances of previous versions of the persistent classes are stored can no longer be read.

In order to overcome this, the configuration manager determines the version of the components of the master database and operates accordingly. Where the version is older than that currently in use, it preferably takes the necessary steps to convert the components. It may also issue some warning to the user. On the other hand, when the components of the master database are from a more recent version than that currently in use, the configuration manager may merely have to issue a warning to the user and prevent use of those components.

Considering a more specific example of the operation, write and read functions write and read a persistent class's data members, both static and non-static to and from an input/output stream. This approach, however, may cause problems when class definitions are changed as illustrated in Figure 7, which shows two versions of a class A.

The pseudo-code shows two class definitions with corresponding read and write functions for the persistence. The instance of class A, written with the first implementation, cannot be read with the second implementation. In particular, the second implementation will try to read a float, but will encounter an integer.

To overcome this, it is proposed that version information is written out together with the persistent classes. This version information relates to the version of the implementation that was used to write the class.

Each persistent class contains a different read function for each (past) version that must still be supported by the robust persistence mechanism. Each read function reads the values of the corresponding old implementation and coverts them to the values needed by the current implementation. Applied to the example of class A, this results in the pseudo-code illustrated in Figure 8.

When an instance is read, the read function that is used to read the instance is selected from the version information found in the stream. When an instance is read to a stream, it is always written with the current (and latest) implementation so that the write method is not version-based.

Thus, the configuration manager may provide a framework that enables classes to register read functions on a version basis and that is able to select the correct read function, based on the version information found in the stream. The implementation of this framework is centred around three classes. The *PersistenceVersionInfo* class encapsulates the version information, such as major and minor version number and status such as alpha, beta or release.

The association between the different versions and their corresponding read functions is then managed by the *PersistenceClassIOManager*. The core of the robust persistence is the *PersistenceManager* class, that maintains a *PersistenceClassIOManager* class for each persistent class.

Considering again the four layer structure illustrated in Figure 6, the persistence layer is thus provided in the first layer.

The second layer comprises the core classes. The common code base consists of a polymorphic hierarchy of attribute type and value classes, classes that model domain class metadata (i.e. type descriptions), a base class that maintains a set of attribute values, the main classes that derive from this class and enable the properties of components to be changed as discussed above, classes that help manage the object repositories, classes that provide an infrastructure for allowing different views of

objects in the repositories so as to allow collaboration between different object repositories as discussed above and helper classes.

The first and second layers comprising the common code base and persistent layer together make up the *backend* of the configuration manager.

The third layer containing the configuration manager specific classes implement the configuration managers logic as discussed above. In particular, the configuration manager determines which components are required by selected components and takes appropriate action.

As discussed above, to function properly some components of an image may require the presence of other components. For example when a developer adds an application object that runs on a particular metaspace, that metaspace and all the objects that comprise it are required.

Preferably, these relationships are modelled by means of a *dependencies* property defined for different sub classes of the *attributed entity* bases class to be described below. Dependencies are described as the list of objects, packages and metaspaces that a component requires. The logic layer provides the necessary functionalities to analyse these dependencies and to add any missing component to the image. The components that make up the basic operating system have their dependencies pre configured in the master repository. However, application developers may also specify dependencies through the graphical user interface (GUI).

Finally, the fourth layer comprises the graphical user interface (GUI) and concerns itself with presentation and input verification using APIs provided by the layers two and three.

CLAIMS

1. A configuration manager for use in forming an image from a plurality of components including objects and metaspaces, the configuration manager including:

a selector for selecting components from a database for use in the image;

an analyser for analysing selected components to determine what other components are required by said selected components; and

means for ensuring that all components required by the selected components are themselves selected.

2. A configuration manager according to claim 1 wherein:

each component is stored in association with a plurality of attributes, the attributes at least indicating any other components required respectively by said each component and the analyser referring to said attributes.

3. A configuration manager according to claim 1 or 2 wherein:

the means for ensuring includes means for prompting a user to select other components required by the selected components.

4. A configuration manager according to claim 1, 2 or 3 wherein:

the means for ensuring includes an automatic selector for selecting other components required by the selected components.

5. A configuration manager according to any preceding claim further including:

an editor for editing selected components and storing the edited components separately to the database.

6. A configuration manager according to claim 5 further including:

a project file for recording the selection of components as pointers to selected components of the database and for storing edited components.

7. A configuration manager according to claim 5 or 6 when appendant on claim 2 wherein:

the editor is for amending the attributes stored in association with the components such that the properties of a component can be changed without the need for recompilation.

8. A configuration manager according to any preceding claim further including:

a display for displaying all of the components available from the database for selection and all of the components selected for the image.

9. A configuration manager according to any preceding claim further including:

a version controller for determining the version of software used to create the components of the database and, where the version is older than a current version for which the configuration manager was intended, for converting the contents of the components to the current version and storing the converted components.

10. A configuration manager according to any preceding claim comprising a four layer structure including a persistence layer, a common code base, a layer of platform independent core classes and a platform specific graphical user interface layer.

11. A configuration manager according to claim 10 wherein version information is contained in the persistence layer.

12. A configuration manager according to claim 10 or 11 when appendant on claim 2 wherein the attributes are contained in the common code base.

13. A configuration manager according to claim 10, 11 or 12 wherein the means for ensuring is contained in the layer of platform independent core classes.

14. A system for forming an image including a configuration manager according to any preceding claim and a compiler for compiling all of the selected components together to form the image.

15. A method of forming an image from a plurality of components including objects and metaspaces, the method including:

selecting components from a database for use in the image;

analysing selected components to determine what other components are required by said selected components; and

ensuring that all components required by the selected component are themselves selected.

16. A method according to claim 15 further including:

compiling all of the selected components together to form the image.

17. A configuration manger constructed and arranged substantially as hereinbefore described with reference to and as illustrated by Figures 1 to 8 of the accompanying drawings.

18. A method of forming an image substantially as hereinbefore described with reference to and as illustrated by Figures 1 to 8 of the accompanying drawings.

19. A computer readable storage medium having recorded ~~th~~ereon code components that, when loaded on a computer and executed will cause that computer to operate according to any preceding claim.

INVESTOR IN PEOPLE

Application No:	GB 0115891.4	**Examiner:**	Richard Baines
Claims searched:	1 - 19	**Date of search:**	18 March 2002

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.T): G4A (AFL, APL)

Int Cl (Ed.7): G06F 9/44, 9/445

Other: Online: keywords in EPODOC, WPI, JAPIO, INSPEC

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
X	US 5,845,119 (HITACHI) - abstract & figure	1 - 5, 15 & 19
A	US 6,223,342 (GEORGE)	
A	A CASE-oriented configuration management agent, Tuppa & Selberherr, IASTED International Conference, Honolulu, 19-21 Aug 1996, pages 368-371, ISBN 0-88986-211-7.	

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.